🌱 The Most Used Spring Annotations

✅ Component Annotations

Used to define Spring-managed beans in your application.

- **`@Component`** – General-purpose stereotype annotation. Spring detects it during component scanning and manages its lifecycle as a bean.
- **`@Service`** – Specialization of `@Component`, intended for service-layer classes containing business logic.
- **`@Repository`** – Indicates a data access object (DAO). Also enables exception translation, converting DB-specific exceptions into Spring's `DataAccessException`.
- **`@Controller`** – Marks a class as a web controller in Spring MVC. It handles HTTP requests and returns views.
- **`@RestController`** – Combines `@Controller` and `@ResponseBody`, meaning every method returns data (like JSON/XML) instead of a view.

⚙️ Configuration Annotations

Define beans, control scanning, and manage environment setup.

- **`@Configuration`** – Marks a class that contains `@Bean` definitions. Spring uses it to generate bean definitions and service requests.
- **`@Bean`** – Declares a bean to be managed by Spring. Used in methods inside a `@Configuration` class.
- **`@ComponentScan`** – Tells Spring where to scan for components (classes annotated with `@Component`, `@Service`, etc.).
- **`@PropertySource`** – Loads properties from a `.properties` file into Spring's Environment.
- **`@Value`** – Injects values from property files or expressions into fields or methods.
- **`@Import`** – Allows importing additional configuration classes.
- **`@Profile`** – Conditionally registers a bean based on the active Spring profile (`dev`, `test`, `prod`, etc.).
- **`@Conditional`** – Register a bean only if a specific condition is met (e.g., presence of a class or property).
- **`@Lazy`** – Defers bean initialization until it's actually needed. Great for performance optimization.
- **`@DependsOn`** – Specifies bean initialization order by declaring dependencies.

- **`@Primary`** – When multiple beans of the same type exist, this one gets autowired by default.
- **`@Order`** – Controls the order in which components are applied or loaded (e.g., filters or interceptors).
- **`@Scope`** – Defines bean scope: `singleton` (default), `prototype`, `request`, `session`, etc.
- **`@Qualifier`** – Helps resolve conflict when multiple beans of the same type are available by specifying which one to inject.

🤝 Dependency Injection Annotations

- **`@Autowired`** – Automatically injects a dependency by type. Can be applied to constructors, fields, or setters.
- **`@Resource`** – JSR-250 standard. Injects by name (first), then by type.
- **`@Inject`** – JSR-330 standard. Functions like `@Autowired`, but without Spring-specific options like `required=false`.
- **`@Required`** – Ensures that a property must be set in Spring config. Throws error if not initialized (now deprecated in favor of constructor injection).

🚀 Spring Boot Annotations

- **`@SpringBootApplication`** – Combines `@Configuration`, `@EnableAutoConfiguration`, and `@ComponentScan` in a single annotation to bootstrap a Spring Boot app easily.
- **`@EnableAutoConfiguration`** – Tells Spring Boot to auto-configure your application based on the dependencies present on the classpath.
- **`@ConfigurationProperties`** – Binds external configuration (like from `application.properties`) to a POJO and validates them using JSR-303/JSR-380.
- **`@ConditionalOnClass`** – Loads a bean or configuration only if a specified class is available in the classpath.
- **`@ConditionalOnMissingClass`** – Opposite of `@ConditionalOnClass`; activates if the class is not present.
- **`@ConditionalOnBean`** – Loads configuration or beans only if a certain bean exists.
- **`@ConditionalOnMissingBean`** – Loads the bean only if the specified bean is *not* present in the context.
- **`@ConditionalOnProperty`** – Activates beans/config based on a property's presence and value.
- **`@EnableConfigurationProperties`** – Enables support for `@ConfigurationProperties`-annotated beans.
- **`@ConstructorBinding`** –Indicates that configuration properties should be bound using the constructor instead of setters.

- **`@ConfigurationPropertiesScan`** – Automatically scans for `@ConfigurationProperties` annotated beans.

🌐 Spring MVC Annotations

- **`@RequestMapping`** – Maps HTTP requests to handler methods; can be used at class or method level.
- **`@GetMapping / @PostMapping / @PutMapping / @DeleteMapping / @PatchMapping`** – Shorthand for `@RequestMapping` for specific HTTP methods.
- **`@PathVariable`** –Binds a URI template variable (e.g., `/users/{id}`) to a method parameter.
- **`@RequestParam`** – Binds a request parameter (e.g., `?name=John`) to a method parameter.
- **`@RequestBody`** – Automatically deserializes JSON/XML request body into a Java object.
- **`@ResponseBody`** – Serializes the return value of a method directly into the response body (often JSON).
- **`@ResponseStatus`** – Specifies the HTTP status code returned from a method or exception.
- **`@ExceptionHandle`** – Defines a method to handle exceptions thrown by controller methods.
- **`@ControllerAdvice`** – Centralized error handling for all controllers in the application.
- **`@RestControllerAdvice`** – Combines `@ControllerAdvice` with `@ResponseBody` for REST APIs.
- **`@SessionAttributes`** – Declares session-scoped model attributes.
- **`@ModelAttribute`** – Binds a method parameter or return value to a model attribute, or initializes it before a controller method.
- **`@InitBinder`** – Initializes data binders for specific fields or parameter types.
- **`@CookieValue`** – Binds method parameters to HTTP cookie values.
- **`@RequestHeader`** – Binds method parameters to HTTP header values.
- **`@CrossOrigin`** – Enables CORS on handler methods or controller classes.

🔐 Spring Security Annotations

- **`@EnableWebSecurity`** – Enables Spring Security's web security support and provides a configuration hook for customizing security behavior.
- **`@Secured`** – Restricts access to a method by specifying required roles (e.g., `@Secured("ROLE_ADMIN")`).
- **`@PreAuthorize`** – Checks authorization *before* method execution using SpEL (Spring Expression Language). Example: `@PreAuthorize("hasRole('ADMIN')")`.

- **`@PostAuthorize`** – Runs after method execution to perform authorization checks on the returned object.
- **`@RolesAllowed`** – Standard Java (JSR-250) way to define allowed roles for accessing a method.
- **`@PreFilter`** – Filters elements of a collection before the method is executed, based on a SpEL condition.
- **`@PostFilter`** – Filters the returned collection based on a SpEL expression after method execution.
- **`@AuthenticationPrincipal`** – Injects the current authenticated principal (user) into a controller method.
- **`@CurrentSecurityContext`** – Provides access to the Spring Security context within methods.
- **`@EnableGlobalMethodSecurity`** – Enables method-level security annotations like `@PreAuthorize`, `@Secured`, etc.
- **`@EnableGlobalAuthentication`** – Allows global configuration of authentication settings (used internally by Spring).

💾 Spring Data / JPA Annotations

- **`@Entity`** – Marks a class as a persistent JPA entity, mapped to a database table.
- **`@Table`** – Specifies the database table name and attributes for the entity.
- **`@Id`** – Indicates the primary key of an entity.
- **`@GeneratedValue`** – Specifies how the primary key is generated (auto-increment, UUID, etc.).
- **`@Column`** – Maps a field to a specific database column with optional constraints.
- **`@Transient`** – Prevents a field from being persisted in the database.
- **`@Temporal`** – Specifies the precision (DATE, TIME, TIMESTAMP) for date/time fields.
- **`@Enumerated`** – Defines how enum fields are persisted (as a string or ordinal).
- **`@Lob`** – Maps a field to a large object column (e.g., BLOB or CLOB).
- **`@OneToOne / @OneToMany / @ManyToOne / @ManyToMany`** – Define relationships between entities.
- **`@JoinColumn`** – Specifies the foreign key column in relationships.
- **`@JoinTable`** – Defines a join table for many-to-many associations.
- **`@Query`** – Writes custom JPQL/SQL queries directly in repository interfaces.
- **`@NamedQuery`** – Declares a named JPQL query at the entity level.
- **`@Modifying`** – Marks a query method that changes the database state (insert/update/delete).
- **`@Transactional`** – Defines transactional boundaries; can be used at method or class level.

- **`@NoRepositoryBean`** – Indicates that the interface is not to be instantiated as a repository bean.
- **`@Param`** – Names parameters used in `@Query` statements.
- **`@EntityListeners`** – Specifies entity lifecycle event listeners.
- **`@CreatedDate / @LastModifiedDate`** – Auto-fill fields with creation and last modification timestamps.
- **`@CreatedBy / @LastModifiedBy`** – Auto-fill fields with user info when creating or modifying an entity.

Spring Repository Annotations

- **`@Repository`** – Marks a class as a Data Access Object (DAO). It is a specialization of `@Component` and is automatically detected during component scanning. Also, it enables automatic translation of persistence-related exceptions into Spring's `DataAccessException`.
- **`@Query`** – Used in Spring Data repositories to define custom JPQL or native SQL queries directly on repository methods.
- **`@Modifying`** – Applied to repository methods that perform modifying queries (such as `UPDATE` or `DELETE`). Must be used along with `@Query`.
- **`@NoRepositoryBean`** – Indicates that a repository interface should not be instantiated directly. Useful when defining base repository interfaces that other interfaces will extend.
- **`@Param`** – Used to name parameters in a custom `@Query` so they can be referred to by name in the query.
- **`@EnableJpaRepositories`** – Enables scanning for Spring Data JPA repositories and configures them.

Spring Transaction Annotations

- **`@Transactional`** – Declares that a method or class should be executed within a transaction. If an exception is thrown, the transaction can be rolled back automatically. You can apply it at the class level (applies to all methods) or method level individually.

  Key attributes:

  - `propagation`: Determines how transactions relate (e.g., `REQUIRED`, `REQUIRES_NEW`).
  - `isolation`: Defines the isolation level of the transaction (e.g., `READ_COMMITTED`).
  - `readOnly`: Marks the transaction as read-only.
  - `rollbackFor`: Specifies which exceptions trigger a rollback.

- `@EnableTransactionManagement` – Enables Spring's annotation-driven transaction management. Required for `@Transactional` to work unless you're using Spring Boot (which auto-enables it).
- `@Rollback` **(test-specific)** – Used in test classes to indicate whether the transaction should roll back after the test.
- `@Commit` **(test-specific)** – Overrides the default rollback behavior for a test method and forces a commit instead.

💾 Spring Cache Annotations

- `@EnableCaching`: Enables annotation-driven cache management.
- `@Cacheable`: Caches the result of a method call based on parameters.
- `@CachePut`: Updates the cache without interfering with the method execution.
- `@CacheEvict`: Removes entries from the cache.
- `@Caching`: Groups multiple caching annotations on the same method.
- `@CacheConfig`: Provides common cache settings for all methods in a class.

🧪 Spring Testing Annotations

- `@SpringBootTest` – Loads the full application context for integration testing.
- `@WebMvcTest` – Loads only the web layer (controllers) for focused tests.
- `@DataJpaTest` – Sets up JPA repositories for testing with in-memory DB.
- `@MockBean` – Adds a Mockito mock of a bean into the Spring application context.
- `@SpyBean` – Adds a Mockito spy of a bean into the context to verify interactions.
- `@ContextConfiguration` – Customizes the ApplicationContext for tests.
- `@ActiveProfiles` – Activates specific Spring profiles during test execution.
- `@TestPropertySource` – Adds specific property files or inlined properties for tests.
- `@DirtiesContext` – Marks the ApplicationContext as dirty, forcing it to reload for the next test.
- `@Sql` – Executes SQL scripts before or after tests to set up or tear down data.
- `@Rollback` – Indicates that a test transaction should be rolled back after the test.
- `@Commit` – Forces a test transaction to commit instead of rolling back.
- `@TestConfiguration` – Declares test-specific configuration beans (similar to `@Configuration`).

☁️ Spring Cloud Annotations

These annotations help in building microservices using Spring Cloud components like service discovery, config server, load balancing, API gateway, etc.

- **`@EnableDiscoveryClient`** – Enables the service to register with a service discovery system (like Eureka, Consul, or Zookeeper). It makes the app discoverable by other services.
- **`@EnableEurekaClient`** – Specifically enables Eureka-based service registration. It's a more specific form of `@EnableDiscoveryClient` and ties the service to the Netflix Eureka registry.
- **`@EnableCircuitBreaker`** – Enables Circuit Breaker pattern support (Hystrix or Resilience4j) for fault tolerance. It automatically stops calling a failing service and provides fallback logic.

- **`@EnableConfigServer`** – Turns your application into a centralized configuration server, allowing other services to fetch their config properties from a central location (typically via Git).
- **`@EnableFeignClients`** – Enables the use of Feign clients, which are declarative REST clients. It automatically generates REST client implementations from annotated interfaces.
- **`@FeignClient`** – Declares an interface as a Feign client. It simplifies calling REST services using Java method invocations instead of writing boilerplate code using `RestTemplate`.
- **`@HystrixCommand`** – Wraps a method with circuit breaker logic. If the method fails or times out, a fallback method is triggered instead (deprecated in favor of Resilience4j).
- **`@LoadBalanced`** – When placed on a `RestTemplate` bean, enables client-side load balancing. It allows service-to-service communication using the service name rather than hardcoded URLs.
- **`@EnableZuulProxy`** – Enables Zuul as an API Gateway. It provides dynamic routing, monitoring, resiliency, and security to your microservices architecture.
- **`@EnableResourceServer`** – Marks a microservice as a protected resource server that requires OAuth2 access tokens for access.
- **`@EnableAuthorizationServer`** – Enables OAuth2 authorization server capabilities, allowing the app to issue access tokens to clients.

🌀 AOP (Aspect-Oriented Programming) Annotations

AOP in Spring lets you separate cross-cutting concerns (like logging, security, transactions) from business logic using aspects.

- **`@EnableAspectJAutoProxy`** – Enables support for handling components marked with `@Aspect`. It allows Spring to auto-detect and apply AOP proxies using AspectJ-style annotations.
- **`@Aspect`** – Marks a class as an aspect. The class can contain advice (code to be executed) and pointcuts (where to execute that code).

- `@Pointcut` – Defines a reusable expression that matches join points (places in the code where aspects should be applied). Used to target specific methods or packages.
- `@Before` – Declares advice that runs **before** the matched method execution. Great for logging, authentication, or pre-checks.
- `@After` – Declares advice that runs **after** the matched method, regardless of its outcome (success or exception).
- `@AfterReturning` – Runs **after** a method executes **successfully**. It can also capture and manipulate the return value.
- `@AfterThrowing` – Runs **only if** the target method throws an exception. Useful for error logging or alerting.
- `@Around` – The most powerful advice type. It wraps the method execution—can execute code **before and after**, and even skip method execution or modify arguments and results.

⏱️ Asynchronous and Scheduling Annotations

These annotations help in executing tasks asynchronously or on a schedule without blocking the main thread.

- `@EnableAsync` – Enables Spring's asynchronous method execution. When used, any method annotated with `@Async` will run in a separate thread.
- `@Async` – Marks a method to be executed asynchronously. This allows long-running tasks (like sending emails or processing files) to run without blocking the caller.
- `@EnableScheduling` – Activates Spring's scheduled task execution capability. Without this, `@Scheduled` won't work.
- `@Scheduled` – Schedules a method to run at fixed intervals or cron expressions. Can be used for tasks like periodic cleanups or data syncing.
  Example: `@Scheduled(fixedRate = 5000)` or `@Scheduled(cron = "0 0 * * * *")`
- `@Schedules` – Allows multiple `@Scheduled` annotations on a single method, useful when the same method needs to run on different schedules.

📬 Messaging Annotations

Spring provides support for messaging systems like **JMS**, **RabbitMQ**, and **Kafka**, and these annotations help define listeners and message mapping logic.

General Messaging Annotations

- `@Payload` – Binds a method parameter to the message payload (the actual data). Works in JMS, Kafka, and RabbitMQ.

- **`@Header`** – Binds a method parameter to a specific message header (like `messageId`, `correlationId`, etc.).
- **`@Headers`** – Injects all message headers into a `Map<String, Object>` parameter for inspection or logic.
- **`@SendTo`** – Specifies where the return value of a listener method should be sent. Used to route responses in message-driven systems.

JMS (Java Message Service)

- **`@EnableJms`** – Enables JMS-related configuration and listener support in Spring applications.
- **`@JmsListener`** – Marks a method as a JMS message listener that gets triggered when a message arrives on a queue/topic.

RabbitMQ

- **`@EnableRabbit`** – Enables RabbitMQ listener support and message-driven POJO configuration.
- **`@RabbitListener`** – Declares a method to listen for messages on RabbitMQ queues.

Kafka

- **`@EnableKafka`** – Enables Kafka listener support in Spring applications.
- **`@KafkaListener`** – Marks a method as a Kafka message listener for a topic or set of topics.